

# Getting Started

with *SchematicSolver* version 2.2

*Miroslav D. Lutovac and Dejan V. Tasic*

## Copyright

---

Miroslav D. Lutovac and Dejan V. Tasic are the holders of the copyright to the *SchematicSolver* software described in this document. ©2003-2009 by Lutovac & Tasic.

## Getting Started Contents

---

<b>Getting Started</b>	1
Copyright	1
Getting Started Contents	2
What is <i>SchematicSolver</i> ?	3
Who Is It For?	4
Overview	5
Features	6
Easy to Use and Learn	6
Powerful Modeling and Simulation Environment	6
Fast and Reliable	7
Implementation of Discrete-Time Systems	8
Teams up with Other <i>Mathematica</i> Applications	8
Documentation	9
Installing <i>SchematicSolver</i>	11
Starting <i>SchematicSolver</i>	12
Registering Your Copy of <i>SchematicSolver</i>	12
Using Online Help	13
<i>SchematicSolver</i> 's Palettes for Interactive Drawing	14
Running Demo	21
Load <i>SchematicSolver</i>	21
Description of Demo System	22
Schematic of Demo System	23
Refining Schematic with Drawing Options	25
Solving Demo System	26
Frequency Response	27
Simulation	29
Software Implementation of Linear System	31
Generating Stimulus	34
Processing with Linear System	35
Nonlinear System	37
Processing with Nonlinear System	39

## What is *SchematicSolver*?

---

Welcome to *SchematicSolver*, a powerful and easy-to-use schematic capture, symbolic analysis, processing, and implementation tool in *Mathematica*. It is a convenient and comprehensive environment in which to draw, solve, simulate, and design systems.

*SchematicSolver* has many unique features not available in other software: symbolic signal processing brings you 1) computation of transfer functions as closed-form expressions in terms of symbolic system parameters, 2) finding the closed-form response from the schematic. The derived result is the most general because all system parameters, inputs, and the initial conditions (states) can be given by symbols.

*SchematicSolver* features automated generation of software implementation of linear and nonlinear discrete systems. The generated implementation function can symbolically process symbolic samples: for a symbolic input sequence, you can compute the symbolic output sequence with both the system parameters and the states specified by symbols.

Other important features include a) design of efficient multirate implementations by working in the symbolic domain, b) modeling systems that work with symbolic complex signals, such as the Hilbert transformer, c) symbolic derivations of important closed-form relations between parameters of a system, such as the power-complementary property of high-speed digital filters, d) symbolic optimization of the system response, and e) functions that generate schematics for arbitrary symbolic system parameters.

*SchematicSolver* can perform signal processing in a traditional numeric way, too.

The *SchematicSolver* application package requires *Mathematica* 7.0 or later. It is developed and supported by Prof. Dr. Miroslav Lutovac and Prof. Dr. Dejan Tomic.

Prof. Dr. Miroslav Lutovac

Bulevar Arsenija Carnojevica 219

11000 Belgrade, Serbia, Europe

**phone:** +381-64-1536934    **fax:** +381-11-137885    **email:** lutovac@etf.bg.ac.rs

<http://www.wolfram.com/products/applications/schematicsolver/>

<http://www.SchematicSolver.com>

## Who Is It For?

---

Whether you are a student, an educator, an engineer, a system analyst, a researcher, or a practitioner, *SchematicSolver* offers you an easy, convenient, and comprehensive environment in which to draw, solve, and implement systems in *Mathematica*.

*SchematicSolver* is targeted at

- educators and students who want more efficient practical teaching and learning
- practitioners who are short of time to master theoretical background of the design procedures, implementation details, processing algorithms, and *Mathematica*
- industry designers responsible for products with short time-to-market
- beginners who learn and experiment with system analysis, implementation, and design
- advanced users who explore and prototype new design algorithms and solutions

You don't have to be a skillful user of *Mathematica*, nor an expert in signal processing, to fully exploit *SchematicSolver*. With just a minimum of basic system theory, you can successfully use *SchematicSolver* to draw, solve, implement, and simulate various systems, such as continuous-time (analog) systems, dynamic feedback and control systems, or discrete-time (digital) multirate systems. *SchematicSolver* has intuitive interface and comprehensive online documentation that leads you step-by-step through the process of creating and analyzing the schematic of your system model.

If you are a signal-processing expert, *SchematicSolver* is a quick solution to your frequently used systems. Moreover, you can use the power of *Mathematica* to full extent for additional processing of symbolic results returned by *SchematicSolver*, such as mixed symbolic-numeric optimization not available with other software.

The interactive online documentation contains a number of detailed real-life examples that demonstrate the use of different schematics – system models – that make *SchematicSolver* an excellent teaching tool either for independent study or for use in Signal Processing, Control Systems, Filter Design, or Signals and Systems courses.

[Go To Contents](#)

## Overview

---

*SchematicSolver* is a *Mathematica* application package for drawing, solving, and implementing systems represented by schematics. It performs mixed symbolic-numeric processing. It is the first mouse-driven interactive drawing tool in *Mathematica*.

Some *SchematicSolver*'s unique drawing features not available in other software follow: a) The graphical representation of a system is not a frozen picture (it is not a bitmap image); it changes automatically when you change system parameters. b) A large schematic, that is the system model, can consist of replicas of simpler schematics; you can write a code to automate drawing for an arbitrary number of repeated parts. c) Functions exist that generate schematics for arbitrary symbolic system parameters.

Using *SchematicSolver* you can perform fast and accurate simulations of discrete-time (digital) and continuous-time (analog) systems, such as velocity servo systems, adaptive LMS systems, automatic gain control (AGC) systems, quadrature amplitude modulation (QAM) systems, square-law envelope detectors, thermodynamics of a house, high-speed recursive filters, Hilbert transformer, efficient multirate systems, dynamic feedback and control systems, digital filters, and nonlinear discrete-time systems.

Symbolic signal processing, a *SchematicSolver*'s unique feature not available in other software, brings you 1) computation of transfer functions as closed-form expressions in terms of symbolic system parameters, 2) finding the closed-form response from the schematic. The derived result is the most general because all system parameters, inputs, and the initial conditions (states) can be given by symbols.

*SchematicSolver* features automated generation of software implementation of linear and nonlinear discrete systems. The generated implementation function can symbolically process symbolic samples: for a symbolic input sequence you can compute the symbolic output sequence with both the system parameters and the states specified by symbols.

Special features include design of efficient multirate implementations by working in the symbolic domain, and modeling systems that work with symbolic complex signals.

*SchematicSolver* is based on the *Mathematica* built-in functions, graphics primitives, and palettes. It is designed for use with *Mathematica* 7.0 or later and for Windows XP/Vista.

## Features

---

### ■ Easy to Use and Learn

- Well-organized palettes for drawing and solving systems by single mouse click
- The first mouse-driven interactive drawing tool entirely based on the *Mathematica* built-in functions, graphics primitives, and palettes
- Powerful functions constructed so that the minimum amount of information has to be specified by the user when modeling or solving a system
- Functions exist that generate schematics for arbitrary symbolic system parameters
- Visualization tools for drawing publication-quality schematics and viewing system models and response
- The graphical representation of a system is not a frozen picture (it is not a bitmap image); it changes automatically when you change system parameters
- Large schematic can consist of replicas of simpler schematics; you can write a code to automate drawing for an arbitrary number of repeated parts
- Extensive online documentation including illustrative application examples and comprehensive reference with Help index
- Requires a minimum understanding of basic system theory and signal processing

### ■ Powerful Modeling and Simulation Environment

- Symbolic signal processing, a *SchematicSolver*'s unique feature not available in other software, brings you computation of transfer functions as closed-form expressions in terms of symbolic system parameters
- Computes transfer function matrix of a multiple-input multiple-output (MIMO) system
- Finds the closed-form response (signals at nodes of the system) directly from the schematic; the derived result is the most general because all system parameters, inputs, and the initial conditions (states) can be given by symbols

Performs fast and accurate simulations of discrete-time (digital) and continuous-time (analog) systems, such as velocity servo systems, adaptive LMS systems, automatic gain control (AGC) systems, quadrature amplitude modulation (QAM) systems, square-law envelope detectors, thermodynamics of a house, high-speed recursive filters, Hilbert transformer, efficient multirate systems, dynamic feedback and control systems, digital filters, and nonlinear discrete-time systems

- Models systems that work with symbolic complex signals, such as the Hilbert transformer
- Carries out symbolic optimization of the system response and mixed symbolic-numeric signal processing
- Performs signal processing in a traditional numeric way

### ■ Fast and Reliable

- By single mouse click symbolically solves, simulates, or implements a system directly from the schematic: a) sets up the equations describing the system, b) computes the system response and transfer functions, c) generates the implementation function
- Helps you generate efficient multirate implementations by working in the symbolic domain
- Provides symbolic derivations of important closed-form relations between parameters of a system, such as the power-complementary property of high-speed digital filters
- Finds closed-form expressions of output signals, for known stimuli given by closed-form expressions, for certain classes of nonlinear systems
- Solves systems with unconnected elements: signals at unconnected element inputs are automatically generated as unique symbols
- Helps you design systems: for known symbolic transfer function, impulse, or step response, you can generate the schematic of the system and find the system parameters

### ■ Implementation of Discrete-Time Systems

- Automated generation of software implementation of linear and nonlinear discrete systems directly from the schematic

The generated implementation function can process symbolic samples one-by-one

- For a symbolic input sequence you can compute the symbolic output sequence with both the system parameters and the initial conditions (states) specified by symbols
- Sets up symbolic implementation equations directly from the schematic
- You can process a list of data samples for a given transfer function; the transfer function is automatically implemented as a single-input single-output Transposed Direct Form 2 IIR discrete system
- Provides functions a) for upsampling and downsampling discrete signals and b) for generating most common discrete signals, such as impulse sequences, step sequences, ramp sequences, sinusoidal or exponential sequences, and random (noise) sequences.
- Includes functions to plot a) frequency response, b) sequences that represent discrete signals, c) Discrete Fourier Transform spectrum, and d) Discrete-Time Fourier Transform spectrum

### ■ Teams up with Other *Mathematica* Applications

- Access to all of the capabilities of *Mathematica* to perform further manipulations on results returned by *SchematicSolver*
- Complements *Control System Professional* with tools for drawing and solving systems described by block-diagrams
- Provides objects, such as symbolic transfer functions, for further analysis by *Signals and Systems Pack*

[Go To Contents](#)

## Documentation

---

*SchematicSolver* has comprehensive online documentation that leads you step-by-step through the process of creating and analyzing your schematic. The interactive online documentation contains a number of detailed examples that demonstrate the use of different schematics – system models.

**Getting Started.** A step-by-step tutorial and quick tour that demonstrates a) how to draw the schematic of a system based on a given physical model, b) how to solve and implement the system model represented by the schematic.

**Introduction.** Main features of *SchematicSolver*, required user background, and technical support.

**Quick Tour of *SchematicSolver*.** A brief description of unique features not available in other software.

**System Representation.** Basic definitions. Discrete, continuous-time, and nonlinear elements. Drawing options. Showing schematics. Signals and transforms.

**Solving Systems.** Discrete and continuous-time linear systems (system equations, response, transfer function, and frequency response). Nonlinear discrete systems.

**Examples of Solving Systems.** Diving submarine system, Unstable plant system, Supply and demand system, Unity feedback system, Satellite elevation tracking system, CD-media controller, Shuttle pitch controller, Direct form 2 transposed IIR filter, State-space model of discrete system. Symbolic optimization of a continuous-time system. Design of a continuous-time system from the step response. Automated drawing and solving of general systems.

**Solving Large Systems.** Combining schematics to build a large system model.

**Implementation of Discrete Systems.** A step-by-step procedure to generate software implementation of systems.

**Nonlinear Discrete System Implementation.** Nonlinear algebraic function element. Nonlinear modulator element. Symbolic solving nonlinear system.

**Examples of Discrete System Implementation.** Adaptive LMS system. Automatic gain control. Quadrature amplitude modulation (QAM). Square-law envelope detector. Nonlinear system with hysteresis. High-speed recursive filters.

**Hilbert Transformer.** Real, complex, and analytic discrete signals. Spectrum of analytic signals. Ideal discrete Hilbert transformer. Processing with Hilbert transformer system. QAM with Hilbert transformer.

**Multirate Systems.** Decimation, Downsampling identity, interpolation, Upsampling identity. Decimation FIR filter. Polyphase decimation. Efficient decimation and interpolation filters. Symbolic multirate processing.

**Hierarchical Systems.** Draw and simulate composite systems. Implementation of hierarchical systems.

**Palettes for drawing and solving systems.** Using palettes. Drawing and editing schematics with palettes. Solving, simulating, and implementing systems with palettes. Drawing large schematics.

**Reference.** Alphabetic list of all functions, options, and reserved symbols with short description of each.

**Processing with *SchematicSolver*.** Symbolic impulse response. Block processing with initial conditions. Processing signals with noise. Processing for given transfer functions.

**Post-processing using *Mathematica*** built-in functions. Representing signals and systems by formulas and operators. Processing with `ZTransform` and `ListConvolve`.

**Post-processing using *Control System Professional* (CSP).** Drawing and solving state-space models using *SchematicSolver*. Simplifying realizations with *SchematicSolver*. Step-by-step procedure for deriving state-space equations.

**Post-processing using *Signals and Systems Pack*.** General report on systems.

**Literature. Table of Contents. Book Index** with more than 200 terms.

[Go To Contents](#)

## Installing *SchematicSolver*

---

*SchematicSolver* is distributed in compressed form as a file *SchematicSolver.zip*.

*SchematicSolver.zip* is a zip archive that contains the *Mathematica* packages and notebooks for the *SchematicSolver* application.

Follow the basic instructions below to install *SchematicSolver* on your computer.

To install the *SchematicSolver* application, you will need to first determine the folder for the files. In a typical Windows XP installation this folder is located at

C:\Documents and Settings\PeterSmith\Application Data\Mathematica\Applications

Unzip in the Applications directory the archive *SchematicSolver.zip* and make sure that you checked the option "Use folder names" in your archiver utility.

After unpacking *SchematicSolver.zip*, new folders appear, e.g.,

..\Applications\SchematicSolver

..\Applications\SchematicSolver\Documentation

..\Applications\SchematicSolver\FrontEnd

..\Applications\SchematicSolver\Kernel

[Go To Contents](#)

## Starting *SchematicSolver*

---

Start *Mathematica* 7.0 or later. Load *SchematicSolver* with the `Get` command

```
In[1]:= Get["SchematicSolver`"]
```

or the `Needs` command

```
In[1]:= Needs["SchematicSolver`"]
```

*SchematicSolver* is one of many available *Mathematica* applications and is normally installed in a separate directory, `SchematicSolver`, in parallel to other applications. If this has been done at the installation stage, the application package should be visible to *Mathematica* without further effort on your part.

This also makes *SchematicSolver* available:

```
In[1]:= << SchematicSolver`
```

## Registering Your Copy of *SchematicSolver*

---

Help us improve *SchematicSolver* by registering your copy. Knowing who uses *SchematicSolver* helps us focus our development efforts and allows us to continue making the kinds of products that will serve you best.

**Additional benefits of registering** are a) free installation support via email for 30 days, b) free technical support via email for 6 months, and c) automatic notification about *SchematicSolver* upgrades.

**How to register?** Send email to

lutovac@kondor.etf.bg.ac.rs

including your name, license number and postal address.

[Go To Contents](#)

## Using Online Help

---

Pull down the Help menu to get immediate access to the *SchematicSolver's* documentation, examples, Table of Contents, Index, and more in the Help Browser:

1. Click Help ▷ Documentation Center ▷ Installed Add-ons.
2. Click SchematicSolver.
3. Click a subcategory in other columns.

Use the Master Index to find information on a particular *SchematicSolver's* topic:

1. Open the Help Browser to the Master Index.
2. In the text field, start typing a keyword.
3. Press **ENTER**. The window lists all available help.
4. Click a *SchematicSolver* hyperlink. The Help Browser jumps to that topic and displays the information.

[Go To Contents](#)

## ***SchematicSolver's* Palettes for Interactive Drawing**

---

Palettes provide a simple way to access the full range of *SchematicSolver's* drawing and solving capabilities.

The *SchematicSolver's* palettes provide an easy point-and-click interface for performing the most common drawing tasks. However, advanced users might prefer to type and evaluate functions directly. But for users who only want to perform the basic operations, the *SchematicSolver's* palettes provide the simplest alternative.

*SchematicSolver* provides four palettes:

- Palette for drawing and solving continuous-time systems, the **Continuous Elements** palette,
- Palette for drawing, solving, simulating, and implementing discrete systems, the **Discrete Elements** palette,
- Palette for drawing, simulating, and implementing discrete nonlinear systems, the **Discrete Nonlinear** palette, and
- Palette for specifying drawing options and schematic plot range, the **Schematic Options** palette.

If the *SchematicSolver's* palettes are not open, choose them with

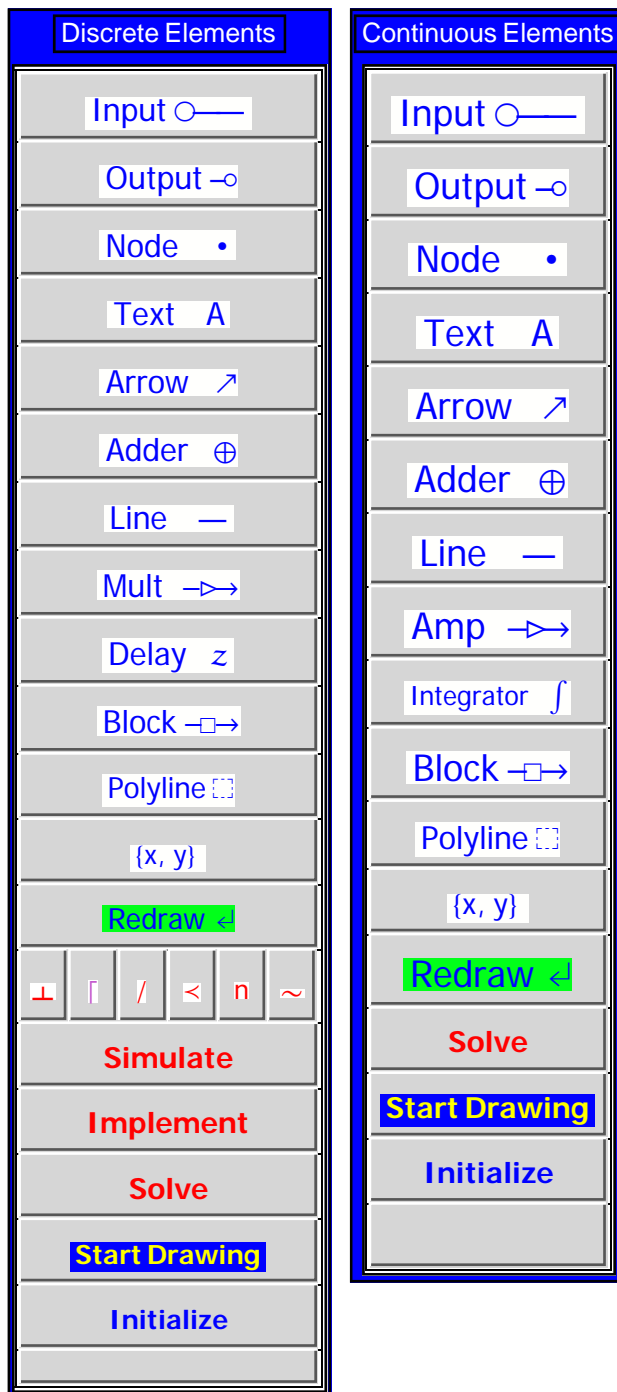
Palettes ▷ DiscreteElements

Palettes ▷ ContinuousElements

Palettes ▷ DiscreteNonlinear

or

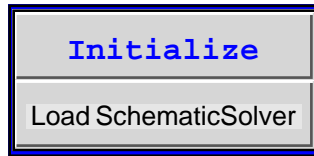
Palettes ▷ SchematicOptions



### To Start Drawing a new Schematic

1. Place the insertion point in a new empty cell in your notebook.

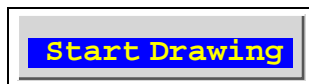
2. Click the button **Initialize** on the palette to load *SchematicSolver*. Palette footer, below the button **Initialize**, indicates the function of this button.



An input cell will be opened with pasted text and then the whole cell will be evaluated:

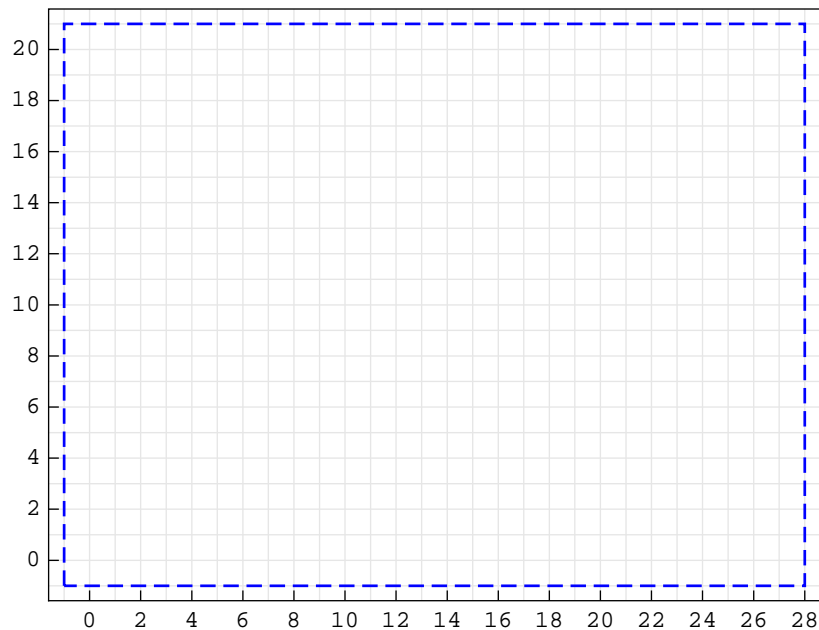
```
Needs["SchematicSolver`"];
SetOptions[InputNotebook[],
  ImageSize -> {350, 300}, WindowSize -> {500, 600}];
```

3. Click the **Start** button



A new input cell will be opened with pasted text. Then the whole cell will be evaluated producing a new graphic output cell below the input cell:

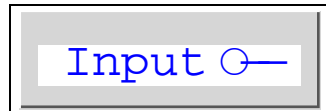
```
In[2]:= mySchematic = {
  {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}};
ShowSchematic[%];
```



By clicking the **Start** button, a new schematic (typically, a system specification) is generated with only one annotation element — Polyline. The `ShowSchematic` function shows the *drawing workspace* with grid lines. By default, the list of elements that describe the schematic is named `mySchematic`. We call this list the *schematic specification*.

4. Place the insertion point in the empty line in your schematic specification, above the drawing workspace.

5. To draw an input, click the **Input** button



Move the mouse over the drawing workspace. Click once, say when the mouse position is over the coordinate `{5, 10}`. The coordinate `{5,10}` is selected, and it appears in the **Input** element specification.

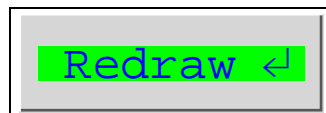
The **Input** element specification is pasted at the current insertion point:

```
{"Input", {5, 10}, x, "", TextOffset → {1, 0}},
```

The schematic specification changes and it has a new element above the empty line.

The insertion point remains in the empty line. The drawing workspace does not change until you evaluate the cell with the schematic specification.

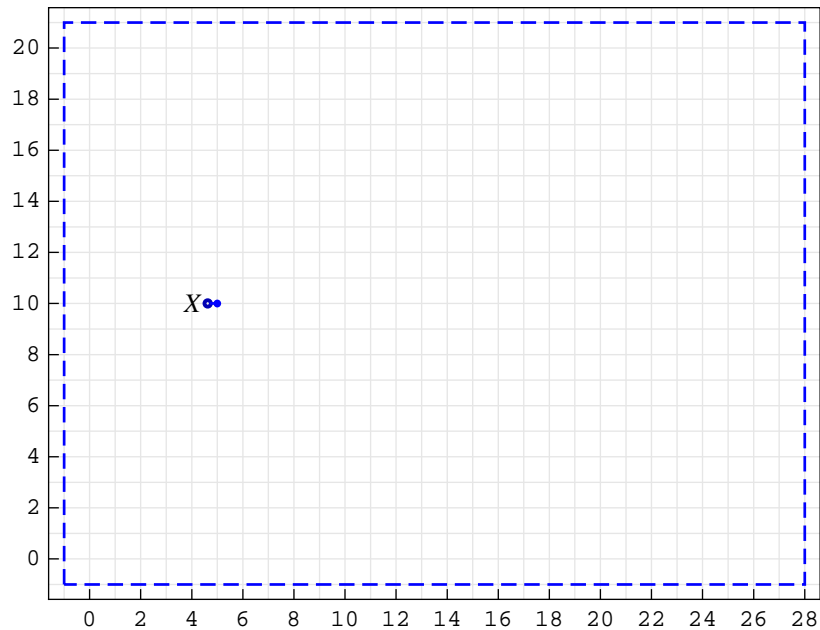
6. Click the **Redraw** button



to redraw the schematic:

```
In[4]:= mySchematic = {
  {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},

  {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}};
ShowSchematic[%];
```



The cell insertion bar appears below the drawing workspace.

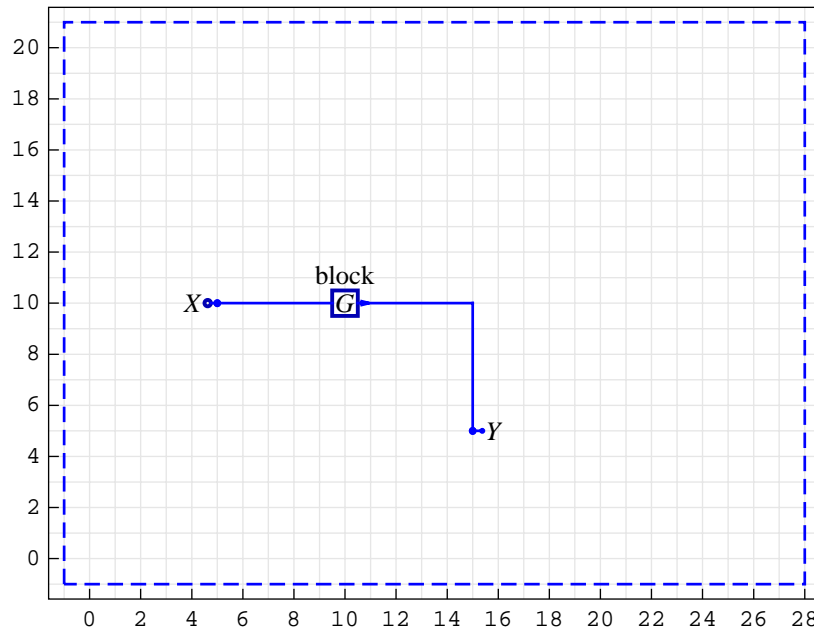
7. Place the insertion point in the empty line in your schematic specification, above the drawing workspace.

8. You can continue filling in your schematic specification with other elements. For example, to add the Block element, click the **Block** button. Move the mouse over the drawing workspace. Press and hold the mouse button, say when the mouse position is over the coordinate {5, 10}. Drag the mouse to specify the second coordinate. Release the mouse button, say at {15, 5}. The schematic specification changes and it has a new element above the empty line.

In a similar way, you can add the Output element at {15, 5}.

Here is the corresponding schematic specification:

```
In[6]:= mySchematic = {  
  {"Input", {5, 10}, X, "", TextOffset -> {1, 0}},  
  {"Block", {{5, 10}, {15, 5}}, G, "block"},  
  {"Output", {15, 5}, Y, "", TextOffset -> {-1, 0}},  
  
  {"Polyline", {{-1, -1}, {-1, 21}, {28, 21}, {28, -1}, {-1, -1}}};  
ShowSchematic[%];
```



Typically, we want to solve the system: to find the system response, or to compute the transfer function. The palette button **Solve** pastes and evaluates a template for general solving a system. The **Solve** button assumes that the name of the schematic specification is `mySchematic`:

```

In[8]:= Print["Equations of the System:"];
        {myEquations, myVars} = DiscreteSystemEquations[mySchematic];
        myEquations // Column
        Print["Response of the System:"];
        {myResponse, myVars} = DiscreteSystemResponse[mySchematic];
        myResponse // Column
        Print["Signals of the System:"];
        {mySignals, myVars} = DiscreteSystemSignals[mySchematic];
        % // Transpose // TableForm
        Print["Transfer Function Matrix:"];
        {myTF, myInputs, myOutputs} =
          DiscreteSystemTransferFunction[mySchematic];
        myTF // MatrixForm
        Print["Inputs of the System:"];
        myInputs
        Print["Outputs of the System:"];
        myOutputs

```

Equations of the System:

```

Out[10]= Y[{5, 10}] = X
         Y[{15, 5}] = G Y[{5, 10}]

```

Response of the System:

```

Out[13]= Y[{15, 5}] → G X
         Y[{5, 10}] → X

```

Signals of the System:

```

Out[16]//TableForm=
      GX Y[{15, 5}]
      X  Y[{5, 10}]

```

Transfer Function Matrix:

```

Out[19]//MatrixForm=
      ( G )

```

Inputs of the System:

```

Out[21]= {Y[{5, 10}]}

```

Outputs of the System:

```

Out[23]= {Y[{15, 5}]}

```

Further processing can be applied to the results returned by `Solve`, say by using *Control System Professional*.

## Running Demo

---

### ■ Load *SchematicSolver*

This makes *SchematicSolver* available:

```
In[24]:= Needs["SchematicSolver`"];
```

We specify some options to better present a demo system:

```
In[25]:= SetOptions[InputNotebook[],  
  ImageSize → {350, 250},  
  ImageMargins → {{0, 0}, {0, 0}}];
```

```
In[26]:= SetOptions[ShowSchematic,  
  ElementScale → 1,  
  FontSize → Automatic,  
  Frame → True,  
  GridLines → Automatic,  
  PlotRange → All];
```

```
In[27]:= SetOptions[DrawElement, ElementSize → {1, 1}, PlotStyle →  
  {{RGBColor[0, 0, 0.7`], Thickness[0.005`], PointSize[0.012`]},  
  {RGBColor[0, 0, 1], Thickness[0.0035`], PointSize[0.01`]}},  
  ShowArrowTail → True, ShowNodes → False, TextOffset → Automatic,  
  BaseStyle → {FontFamily → Times, FontSize → 10}];
```

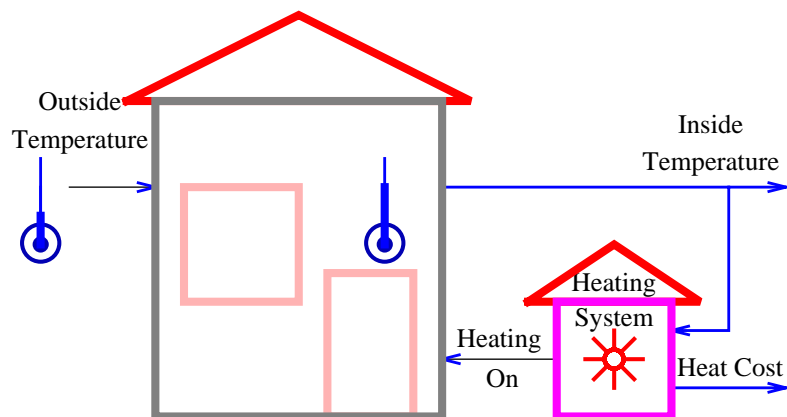
```
In[28]:= SetOptions[SequencePlot,  
  StemPlot → False,  
  Joined → True];
```

[Go To Contents](#)

## ■ Description of Demo System

Let us consider a simple model of the thermodynamics of a house. Here is the system linear created with *SchematicSolver*:

```
In[29]:= ShowSchematic[
  SchematicSolverFigureImplementationExamplesHouseHeating,
  GridLines -> None, Frame -> False]
```



The out-door thermometer measures the outside temperature, `tempOut`, and the in-door thermometer measures the inside temperature, `tempIn`. The temperatures have been obtained by taking samples at discrete instants of time. We are concerned with uniform samples by sampling every  $T$  units of time.

The next sample of the inside temperature is obtained by adding two terms to the current sample of the inside temperature `tempIn`:

$$(\text{tempOut} - \text{tempIn}) * \text{coefHouse}$$

and

$$\text{heatOn} * \text{coefHeat}$$

`coefHouse` denotes a parameter of the house, `coefHeat` denotes a parameter of the heating system, and `heatOn` can be 1 (heating system turned on) or 0 (heating system turned off). The next sample of the cumulative heating cost is computed by adding `unitCost` to the cumulative heating cost if the heating system is turned on.

## ■ Schematic of Demo System

The schematic of the demo system can be drawn according to the system description.

- First, we use the **Input** element to describe the outside temperature `tempOut`.
- We employ the **Adder** element to perform the operation of subtraction of the outside temperature and the inside temperature.
- The difference of those two temperatures is multiplied by `coefHouse` using the **Multiplier** element.
- Another **Input** element is used for `heatOn`.
- The **Multiplier** element is used for multiplying `heatOn` by `coefHeat`.
- The **Adder** element sums the outputs of the multipliers. This value is added to the current inside temperature `tempIn` by using another **Adder** element.
- The computed value becomes the next sample of the inside temperature, therefore we use the **Delay** element.
- The output of the Delay element is fed back to the adders as the value of the current inside temperature – it is represented by the **Output** element.
- The value of `heatOn` is multiplied by `unitCost` using the **Multiplier** element.
- This value is added to the current cumulative heating cost by using the **Adder** element.
- The computed cost becomes the next sample of the cumulative heating cost, therefore we use another **Delay** element.
- The output of the delay element has the value of the current cumulative heating cost – it is represented by the **Output** element.

*SchematicSolver* describes a system as a list of elements; this list specifies what elements are in the system and how they are interconnected. A list describing a system will be referred to as the *system specification*. Each element in the system is also described as a list that states what the element is, to which other elements it is connected, and what its value is. A list describing an element will be referred to as the *element specification*.

Here is the schematic specification of the thermodynamics of a house:

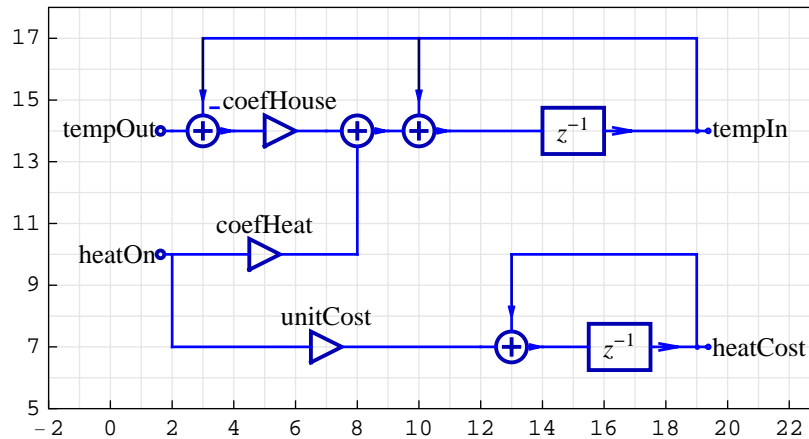
```
In[30]:= heatingSchematic = {
  {"Adder", {{2, 14}, {3, 13}, {4, 14}, {10, 17}}, {1, 0, 2, -1}},
  {"Adder", {{7, 14}, {8, 10}, {9, 14}, {8, 17}}, {1, 1, 2, 0}},
  {"Adder", {{9, 14}, {9, 10}, {11, 14}, {10, 17}}, {1, 0, 2, 1}},
  {"Adder", {{12, 7}, {13, 6}, {14, 7}, {13, 10}}, {1, 0, 2, 1}},
  {"Multiplier", {{4, 14}, {7, 14}}, coefHouse},
  {"Multiplier", {{2, 10}, {8, 10}}, coefHeat},
  {"Multiplier", {{2, 7}, {12, 7}}, unitCost},
  {"Delay", {{11, 14}, {19, 14}},
  1, "", ElementSize → {2, 3 / 2}},
  {"Delay", {{14, 7}, {19, 7}}, 1, "", ElementSize → {2, 3 / 2}},
  {"Line", {{10, 17}, {19, 17}, {19, 14}}},
  {"Line", {{2, 7}, {2, 10}}},
  {"Line", {{13, 10}, {19, 10}, {19, 7}}},
  {"Arrow", {{10, 15}, {10, 17}}},
  {"Arrow", {{3, 15}, {3, 17}}},
  {"Arrow", {{13, 8}, {13, 10}}}};
```

```
In[31]:= heatingInOut = {
  {"Input", {2, 14}, tempOut},
  {"Input", {2, 10}, heatOn},
  {"Output", {19, 14}, tempIn},
  {"Output", {19, 7}, heatCost, "", TextOffset → {-1, 0}}};
```

```
In[32]:= linearHeatingSystem = Join[heatingSchematic, heatingInOut];
```

ShowSchematic shows the system schematic:

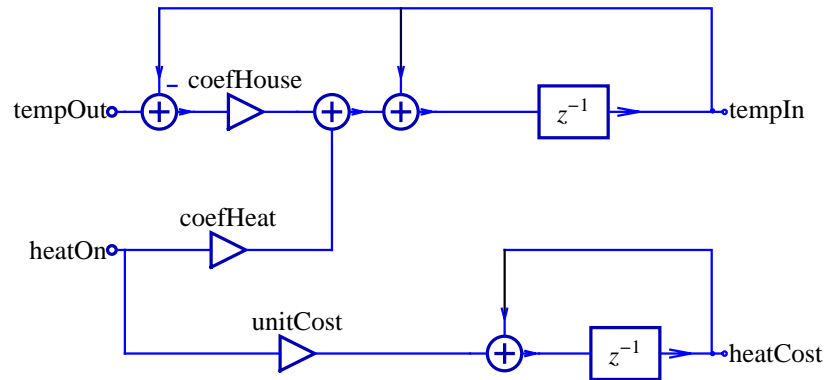
```
In[33]:= ShowSchematic[linearHeatingSystem,
  PlotRange → {{-2, 23}, {5, 18}}];
```



## ■ Refining Schematic with Drawing Options

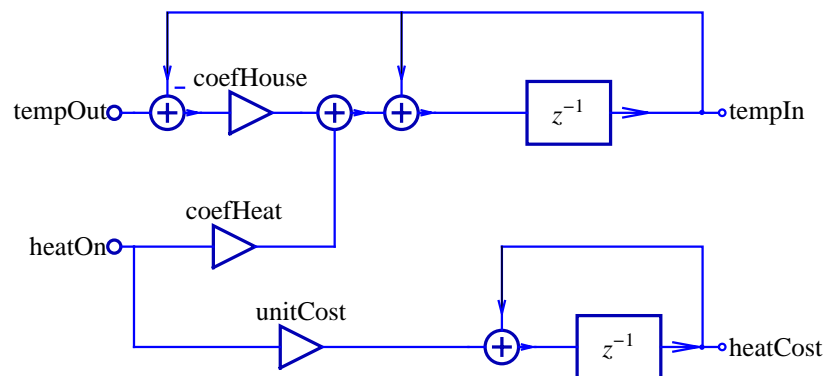
Schematic can be drawn without grid and frame:

```
In[34]:= ShowSchematic[linearHeatingSystem,
  Frame -> False, GridLines -> None];
```



You can increase the size of all elements by 25% using the option `ElementScale -> 1.25`:

```
In[35]:= ShowSchematic[linearHeatingSystem,
  Frame -> False, GridLines -> None, ElementScale -> 1.25];
```



[Go To Contents](#)

## ■ Solving Demo System

`DiscreteSystemEquations` sets up the equations directly from the schematic:

```
In[36]:= {heatingEqns, vars} =
  DiscreteSystemEquations[linearHeatingSystem];
  heatingEqns // Column

Y[{4, 14}] == Y[{2, 14}] - Y[{10, 17}]
Y[{9, 14}] == Y[{7, 14}] + Y[{8, 10}]
Y[{11, 14}] == Y[{9, 14}] + Y[{10, 17}]
Y[{14, 7}] == Y[{12, 7}] + Y[{13, 10}]
Y[{7, 14}] == coefHouse Y[{4, 14}]
Y[{8, 10}] == coefHeat Y[{2, 7}]
Out[37]= Y[{12, 7}] == unitCost Y[{2, 7}]
  Y[{10, 17}] ==  $\frac{Y[{11,14}]}{z}$ 
  Y[{13, 10}] ==  $\frac{Y[{14,7}]}{z}$ 
  Y[{2, 14}] == tempOut
  Y[{2, 7}] == heatOn
```

`DiscreteSystemTransferFunction` finds the transfer function:

```
In[38]:= {tfMatrix, systemInps, systemOuts} =
  DiscreteSystemTransferFunction[linearHeatingSystem]

Out[38]= {{{  $\frac{\text{coefHouse}}{-1 + \text{coefHouse} + z}$ ,  $\frac{\text{coefHeat}}{-1 + \text{coefHouse} + z}$  }, {0,  $\frac{\text{unitCost}}{-1 + z}$  }},
  {Y[{2, 14}], Y[{2, 7}]}, {Y[{10, 17}], Y[{13, 10}]}}
```

The system has two inputs and two outputs, so *SchematicSolver* computes the transfer function matrix

```
In[39]:= tfMatrix // MatrixForm

Out[39]//MatrixForm=

$$\begin{pmatrix} \frac{\text{coefHouse}}{-1 + \text{coefHouse} + z} & \frac{\text{coefHeat}}{-1 + \text{coefHouse} + z} \\ 0 & \frac{\text{unitCost}}{-1 + z} \end{pmatrix}$$

```

Each row of this matrix corresponds to a system output and each column of the matrix corresponds to a system input. The first input corresponds to the first Input element in `linearHeatingSystem`, the second input corresponds to the second Input element in `linearHeatingSystem`, and so on. The same convention applies to the numbering of outputs.

## ■ Frequency Response

Transfer function of the inside temperature with respect to the outside temperature is

```
In[40]:= tempInTF = tfMatrix[[1, 1]]
```

```
Out[40]= 
$$\frac{\text{coefHouse}}{-1 + \text{coefHouse} + z}$$

```

For specific values of system parameters

```
In[41]:= parameterSubstitution = {  
    coefHeat → 1.022,  
    coefHouse → 0.022,  
    unitCost → 0.025}
```

```
Out[41]= {coefHeat → 1.022, coefHouse → 0.022, unitCost → 0.025}
```

the transfer function becomes

```
In[42]:= tempInTFspecific = tempInTF /. parameterSubstitution
```

```
Out[42]= 
$$\frac{0.022}{-0.978 + z}$$

```

and can be displayed in the more convenient form

```
In[43]:= DiscreteSystemDisplayForm[tempInTFspecific]
```

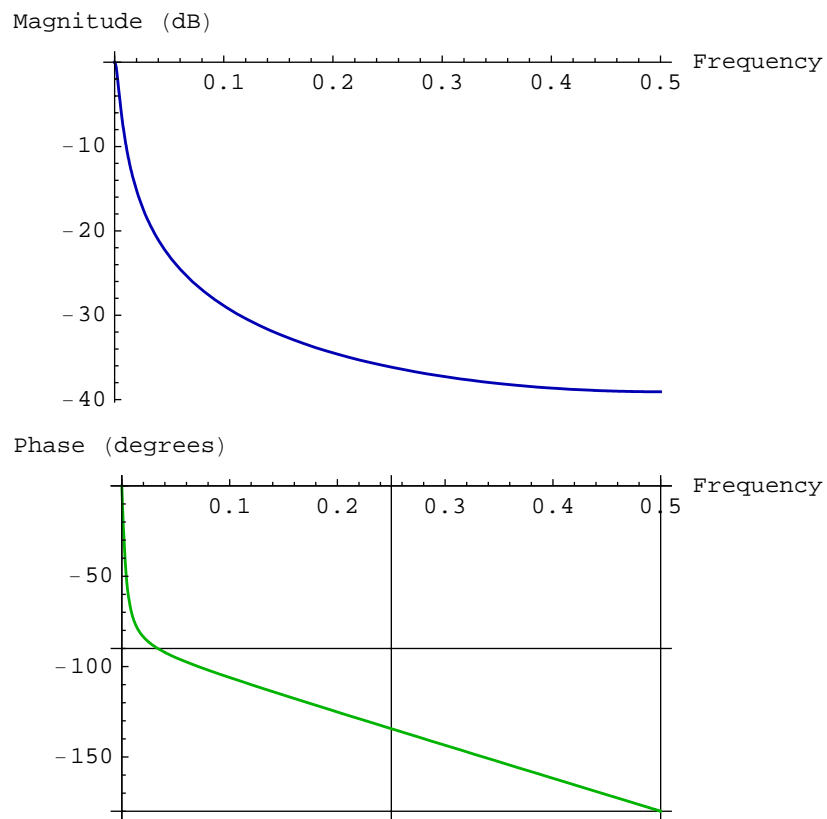
```
Out[43]//DisplayForm=  

$$\frac{0.022 z^{-1}}{1. - 0.978 z^{-1}}$$

```

Here is the frequency response of the system:

```
In[44]:= DiscreteSystemFrequencyResponse[tempIntFspecific];
```



[Go To Contents](#)

## ■ Simulation

Assume that the outside temperature abruptly changes from zero to 70

```
In[45]:= tempOutMax = 70;
```

Here are the first 200 samples of the outside temperature:

```
In[46]:= numberOfSamples = 200;
```

```
In[47]:= inpSeq1 = tempOutMax * UnitStepSequence[numberOfSamples];
```

Assume no heating

```
In[48]:= inpSeq2 = 0 * inpSeq1;
```

MultiplexSequence forms the input sequence to the system:

```
In[49]:= inputSequence = MultiplexSequence[inpSeq1, inpSeq2];
```

For given system parameters

```
In[50]:= parameterSubstitution
```

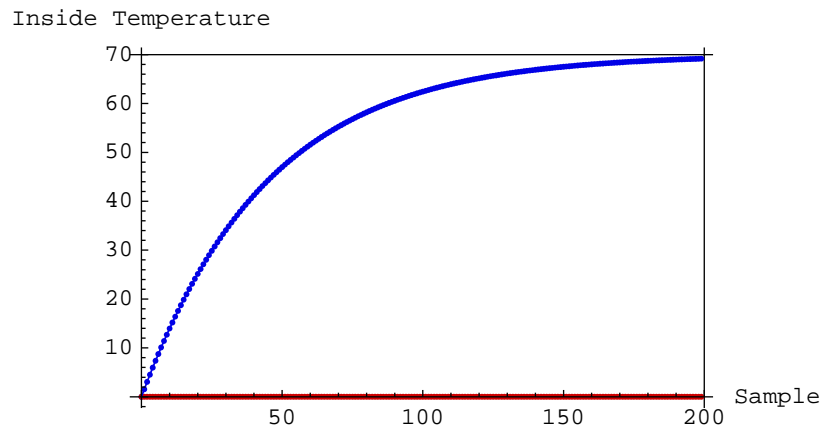
```
Out[50]= {coefHeat → 1.022, coefHouse → 0.022, unitCost → 0.025}
```

DiscreteSystemSimulation finds the system output, for zero initial conditions, as follows

```
In[51]:= outputSequence =  
          DiscreteSystemSimulation[  
            linearHeatingSystem /. parameterSubstitution, inputSequence];
```

SequencePlot plots outputSequence:

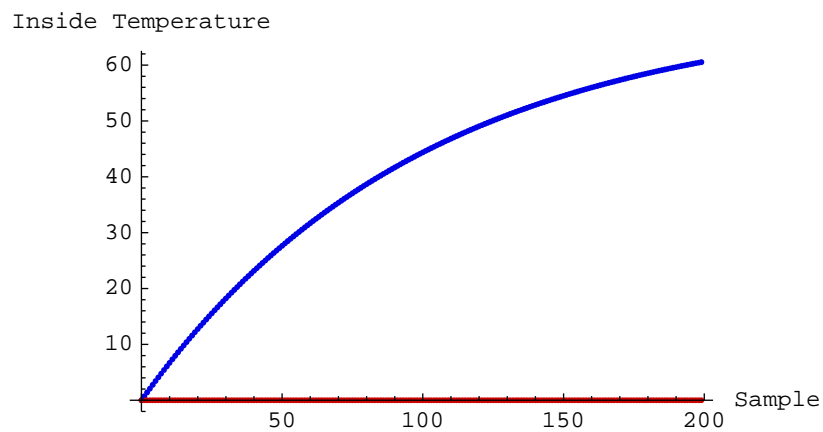
```
In[52]:= SequencePlot[outputSequence,
  AxesLabel → {"Sample", "Inside Temperature"},
  GridLines → {{numberOfSamples}, {tempOutMax}}];
```



After 200 samples, the inside temperature is practically equal to the outside temperature.

You can easily change system parameters on the fly and make a new simulation:

```
In[53]:= outSeq = DiscreteSystemSimulation[linearHeatingSystem /.
  {coefHeat → 1.02, coefHouse → 0.01}, inputSequence];
SequencePlot[%, AxesLabel → {"Sample", "Inside Temperature"}];
```



## ■ Software Implementation of Linear System

*Software implementation* is a sequence of statements that are executed on a general-purpose computer or on a dedicated hardware.

`DiscreteSystemImplementationSummary` points out the system input, initial state, parameter set, output, and final state:

```
In[55]:= DiscreteSystemImplementationSummary[linearHeatingSystem]

Input: {Y[{2, 14}], Y[{2, 7}]}
Initial state: {Y[{10, 17}], Y[{13, 10}]}
Parameter: {coefHeat, coefHouse, unitCost}
Output: {Y[{10, 17}], Y[{13, 10}]}
Final state: {Y[{11, 14}], Y[{14, 7}]}
```

`DiscreteSystemImplementation` creates a *Mathematica* function that implements the system.

```
In[56]:= DiscreteSystemImplementation[
  linearHeatingSystem, "linearSystemImplementation"];
Implementation procedure name: linearSystemImplementation
Implementation procedure usage:
```

```
{{Y10p17, Y13p10}, {Y11p14, Y14p7}}
= linearSystemImplementation[{Y2p14,
Y2p7},{Y10p17, Y13p10},{coefHeat, coefHouse,
unitCost}] is the template for calling the procedure.
The general template is {outputSamples,
finalConditions} = procedureName[inputSamples,
initialConditions, systemParameters]. See
also: DiscreteSystemImplementationProcessing
```

We can use `??` to get full information about the implementation procedure:

```
In[57]:= ?? linearSystemImplementation
```

```
{{Y10p17, Y13p10}, {Y11p14, Y14p7}}
= linearSystemImplementation[{Y2p14,
Y2p7},{Y10p17, Y13p10},{coefHeat, coefHouse,
unitCost}] is the template for calling the procedure.
The general template is {outputSamples,
finalConditions} = procedureName[inputSamples,
initialConditions, systemParameters]. See
also: DiscreteSystemImplementationProcessing
```

```
linearSystemImplementation[] := {2, 2, 3, 11, 2, 2}
```

```
linearSystemImplementation[dataSamples_List,
initialConditions_List, systemParameters_List] :=
Module[{Y2p14, Y2p7, Y10p17, Y13p10, Y4p14, Y7p14, Y8p10,
Y9p14, Y11p14, Y12p7, Y14p7, coefHeat, coefHouse, unitCost},
{coefHeat, coefHouse, unitCost} = systemParameters;
{Y2p14, Y2p7} = dataSamples;
{Y10p17, Y13p10} = initialConditions;
Y4p14 = -Y10p17 + Y2p14; Y7p14 = coefHouse Y4p14;
Y8p10 = coefHeat Y2p7; Y9p14 = Y7p14 + Y8p10;
Y11p14 = Y10p17 + Y9p14; Y12p7 = unitCost Y2p7;
Y14p7 = Y12p7 + Y13p10; {{Y10p17, Y13p10}, {Y11p14, Y14p7}}]
```

DiscreteSystemImplementationEquations is used to extract the system input, initial state, parameter names, implementation equations, output, and final state:

```
In[58]:= {systemInput, initialConditions, parameterNames,
implementationEquations, systemOutput, finalState} =
DiscreteSystemImplementationEquations[linearHeatingSystem];
```

```
In[59]:= systemInput
```

```
Out[59]= {Y[{2, 14}], Y[{2, 7}]}
```

```
In[60]:= initialConditions
```

```
Out[60]= {Y[{10, 17}], Y[{13, 10}]}
```

```
In[61]:= parameterNames
```

```
Out[61]= {coefHeat, coefHouse, unitCost}
```

```
In[62]:= systemOutput
```

```
Out[62]= {Y[{10, 17}], Y[{13, 10}]}
```

```
In[63]:= Column[implementationEquations]
```

```

Y[{2, 14}] = tempOut
Y[{2, 7}] = heatOn
Y[{10, 17}] = previousSample[Y[{11, 14}]]
Y[{13, 10}] = previousSample[Y[{14, 7}]]
Y[{4, 14}] = Y[{2, 14}] - Y[{10, 17}]
Out[63]= Y[{7, 14}] = coefHouse Y[{4, 14}]
Y[{8, 10}] = coefHeat Y[{2, 7}]
Y[{9, 14}] = Y[{7, 14}] + Y[{8, 10}]
Y[{11, 14}] = Y[{9, 14}] + Y[{10, 17}]
Y[{12, 7}] = unitCost Y[{2, 7}]
Y[{14, 7}] = Y[{12, 7}] + Y[{13, 10}]

```

It is better typeset with

```
In[64]:= typoSubstYkn = {Y[{k_Integer, n_Integer}] => Yk,n};
```

```
In[65]:= typoSubst = {coefHouse → αhouse,
coefHeat → γheat, heatOn → Γon, previousSample → D,
systemOuts[[1]] → Θinside, systemOuts[[2]] → Ctotal,
tempOut → Θoutside, unitCost → cunit};
```

```
In[66]:= implementationEquations /. typoSubst /. typoSubstYkn // Column //
TraditionalForm
```

```
Out[66]//TraditionalForm=
```

$$\begin{aligned}
Y_{2,14} &= \Theta_{\text{outside}} \\
Y_{2,7} &= \Gamma_{\text{on}} \\
\Theta_{\text{inside}} &= \mathcal{D}(Y_{11,14}) \\
C_{\text{total}} &= \mathcal{D}(Y_{14,7}) \\
Y_{4,14} &= Y_{2,14} - \Theta_{\text{inside}} \\
Y_{7,14} &= Y_{4,14} \alpha_{\text{house}} \\
Y_{8,10} &= Y_{2,7} \gamma_{\text{heat}} \\
Y_{9,14} &= Y_{7,14} + Y_{8,10} \\
Y_{11,14} &= Y_{9,14} + \Theta_{\text{inside}} \\
Y_{12,7} &= Y_{2,7} c_{\text{unit}} \\
Y_{14,7} &= Y_{12,7} + C_{\text{total}}
\end{aligned}$$

Go To Contents

## ■ Generating Stimulus

We can simulate daily temperature fluctuations applying a sinusoidal term with amplitude of 12 to a base temperature of 55. Assume that we sample temperature every minute, and that we observe an interval of two days.

```
In[67]:= baseTemperature = 55; amplitudeTemperature = 12;
         sinePeriod = 60 * 24; numberOfSamples = 60 * 24 * 2;
```

```
In[69]:= inpSeq1 = baseTemperature + amplitudeTemperature *
          UnitSineSequence[numberOfSamples, 1 / sinePeriod];
```

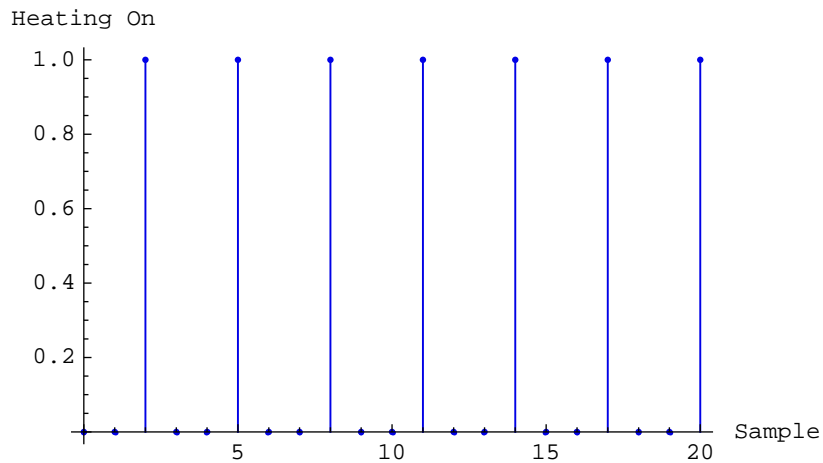
Assume that heating is periodically turned on for 1 minute, and then turned off for 2 minutes:

```
In[70]:= inpSeq2 =
          (1 - Sign[(0.1 + UnitSineSequence[numberOfSamples, (24 * 20) /
          sinePeriod]])) / 2;
```

MultiplexSequence forms the input sequence to the system:

```
In[71]:= inputSequence = MultiplexSequence[inpSeq1, inpSeq2];
```

```
In[72]:= SequencePlot[inpSeq2[[Range[21]]],
          AxesLabel → {"Sample", "Heating On"},
          StemPlot → True, Joined → False];
```



## ■ Processing with Linear System

Assume the following initial conditions (inside temperature of 60 and zero cumulative heating cost):

```
In[73]:= initialSubstitutions =  
        {tempInitialCondition → 60, costInitialCondition → 0};  
        initialConditions = {tempInitialCondition,  
        costInitialCondition} /. initialSubstitutions  
  
Out[74]= {60, 0}
```

Assume the following values for the system parameters:

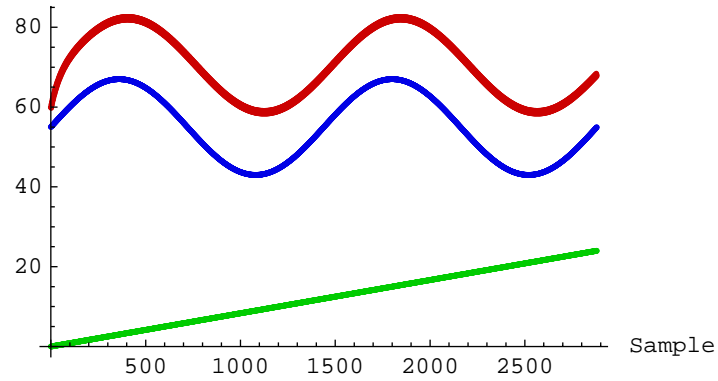
```
In[75]:= parameterSubstitution  
  
Out[75]= {coefHeat → 1.022, coefHouse → 0.022, unitCost → 0.025}  
  
In[76]:= systemParameters = parameterNames /. parameterSubstitution  
  
Out[76]= {1.022, 0.022, 0.025}
```

DiscreteSystemImplementationProcessing processes inputSequence for created linearSystemImplementation.

```
In[77]:= {outputSequence, finalConditions} =  
        DiscreteSystemImplementationProcessing[  
        inputSequence, initialConditions,  
        systemParameters, linearSystemImplementation];
```

```
In[78]:= MultiplexSequence[inpSeq1, outputSequence];
SequencePlot[%,
  AxesLabel →
  {"Sample", "Inside, Outside Temp\n& Cumulative Cost"}];
```

Inside, Outside Temp  
& Cumulative Cost



Outside temperature is plotted in blue, inside temperature is plotted in red, and cumulative heating cost appears in green.

DemultiplexSequence extracts individual output sequences:

```
In[80]:= {insideTemperatureSeq, costSeq} =
  DemultiplexSequence[outputSequence];
```

Here is the final value of the cumulative heating cost:

```
In[81]:= finalCost = Last[costSeq]
```

```
Out[81]= {23.975}
```

[Go To Contents](#)



implements the nonlinear system.

```
In[86]:= DiscreteSystemImplementation[
  nonlinearHeatingSystem, "nonlinearSystemImplementation"];
Implementation procedure name: nonlinearSystemImplementation
Implementation procedure usage:
```

```
{{Y19p12, Y13p10}, {Y11p14, Y14p7}} =
  nonlinearSystemImplementation[{Y2p14}, {Y19p12,
  Y13p10}, {coefHeat, coefHouse, F, unitCost}]
is the template for calling the procedure.
The general template is {outputSamples,
  finalConditions} = procedureName[inputSamples,
  initialConditions, systemParameters]. See
also: DiscreteSystemImplementationProcessing
```

*SchematicSolver* can process symbolic samples for symbolic system parameters:

```
In[87]:= outSymbSeq = DiscreteSystemSimulation[
  nonlinearHeatingSystem /. parameterSubstitution, {{x1}, {x2}}]
Out[87]= {{0, 0}, {0.022 x1 + 1.022 F[0], 0.025 F[0]}}
```

You can assign numeric values to the result after processing:

```
In[88]:= outSymbSeq /. {x1 → 50, F → Cos}
Out[88]= {{0, 0}, {2.122, 0.025}}
```

[Go To Contents](#)

## ■ Processing with Nonlinear System

Consider a function that should keep the inside temperature within a predefined temperature range:

```
In[89]:= tempThermostat = 70;
tempDelta = 5;
tempHeatOn = tempThermostat - tempDelta;
tempHeatOff = tempThermostat + tempDelta;
heatingFlag = 0;

In[94]:= Clear[Fhysteresis];
Fhysteresis[t_] := Module[{heatingSwitch},
  If[heatingFlag == 0,
    If[t < tempHeatOn, heatingFlag = 1;
      heatingSwitch = 1, heatingSwitch = heatingFlag],
    If[t > tempHeatOff, heatingFlag = 0; heatingSwitch = 0,
      heatingSwitch = heatingFlag]];
  heatingSwitch]
```

The function  $F_{\text{hysteresis}}$  uses the global variable `heatingFlag` and changes its value during processing.

If temperature increases from 20 to 90, the heating switch is off after 75. If temperature decreases from 90 to 20, the heating switch is on below 65.

Assume the following initial conditions (inside temperature of 60 and zero cumulative heating cost):

```
In[96]:= initialConditions
Out[96]= {60, 0}
```

The system parameters now contain the function name  $F$ :

```
In[97]:= eqnsObject = DiscreteSystemImplementationEquations[
  nonlinearHeatingSystem];
systemParameters = eqnsObject[[3]] /. parameterSubstitution /.
  F → Fhysteresis

Out[98]= {1.022, 0.022, Fhysteresis, 0.025}
```

The system input is the previously generated stimulus `inpSeq1`.

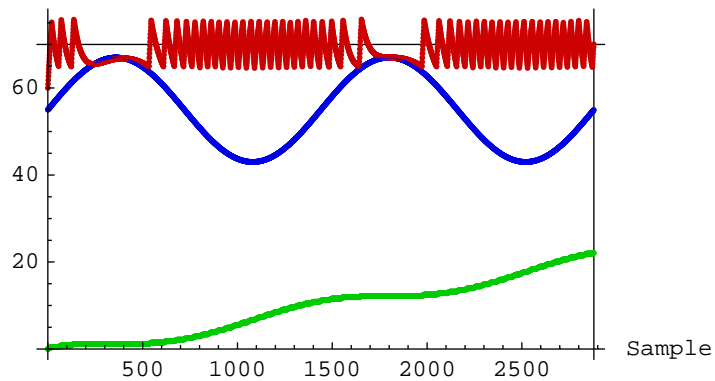
```
In[99]:= inputSequence = inpSeq1;
```

DiscreteSystemImplementationProcessing processes inputSequence for created nonlinearSystemImplementation.

```
In[100]:=
{outputSequence, finalConditions} =
  DiscreteSystemImplementationProcessing[
    inputSequence, initialConditions,
    systemParameters, nonlinearSystemImplementation];

In[101]:=
MultiplexSequence[inpSeq1, outputSequence];
SequencePlot[%,
  AxesLabel →
    {"Sample", "Inside, Outside Temp\n& Cumulative Cost"},
  GridLines → {{numberOfSamples}, {tempThermostat}},
  AxesOrigin → {0, 0}];
```

Inside, Outside Temp  
& Cumulative Cost



Outside temperature is plotted in blue, inside temperature is plotted in red, and cumulative heating cost appears in green.

DemultiplexSequence extracts individual output sequences:

```
In[103]:=
{insideTemperatureSeq, costSeq} =
  DemultiplexSequence[outputSequence];
```

Here is the final value of the cumulative heating cost:

```
In[104]:=
finalCostNonlinear = Last[costSeq]
```

```
Out[104]=
{22.1}
```